Berlin Institute of Technology
FG Security in Telecommunications

OS Agnostic Sandboxing Using Virtual CPUs

Spring 6 - SIDAR Graduierten-Workshop über Reaktive Sicherheit

Matthias Lange, March 21st, 2011
mlange@sec.t-labs.tu-berlin.de

Outline

- Introduction
- Design
- Implementation
- Evaluation
- Conclusion

Introduction

- Insufficient access control mechanisms of current OS
  - No principle of least authority
- Untrusted 3rd party code within trustworthy environment
  - e.g. plugins
- Sandboxing to restrict programs
  - Many different (special purpose) implementations
  - No general approach

Background - Sandboxing

- Jail program into restricted execution environment
- Check adherence to policy
  - Faults trap into sandbox
- Address spaces, Process VMs, Software Fault Isolation
- Java VM
  - Disliked because of performance penalty
- Google NaCL
  - Uses (x86) platform specific features

Design

Design Goals

- Native code execution
  - Performance
- Low complexity
- OS agnostic
- Enable multimedia applications
  - Low latency
  - High data throughput
  - Multiple event sources
- Threading
- Prioritization

Execution Model - Virtual CPUs

- Standard threading model not sufficient
  - Complex upon control flow diversions
- vCPU is an execution abstraction
- Strongly resemble physical CPU
  - Upcalls
  - State indicator
  - Virtual interrupt flag
  - State save area

Host-Client Interaction

- System calls
  - Client state change to notify host
- Events
  - Client notification, upcall to entry point
- State indicator
  - Enable / disable notifications
- State save area
  - Store state of interrupted client thread

Threading Library

- Multi-threading
  - Preemption
  - Scheduling
  - Prioritization of events and threads
  - Synchronisation
- Dynamic memory

Implementation

General Overview

- Linux as host
- Sandboxing implemented using **ptrace**
- vCPU implemented on shared memory page
- Scheduling
  - Fixed priority round robin scheduling
- Event Handling
  - Event handler threads, allows prioritization

VCPU System Calls

- Host waits for client changes using **waitpid**
- Client issues segmentation fault at specific address
- Manipulation of client state using **ptrace**
  - save/restore register state
  - Resume vCPU at entry point vector

Evaluation

Setup

- AMD Athlon 64 X2 dual core @ 2,6GHz
- 3,9 GB RAM
- Ubuntu 9.10

System Call Roundtrip

	Clock cycles	Time in µs
vCPU (syscall_null)	37.702	35.671
native (getpid)	248	0.234

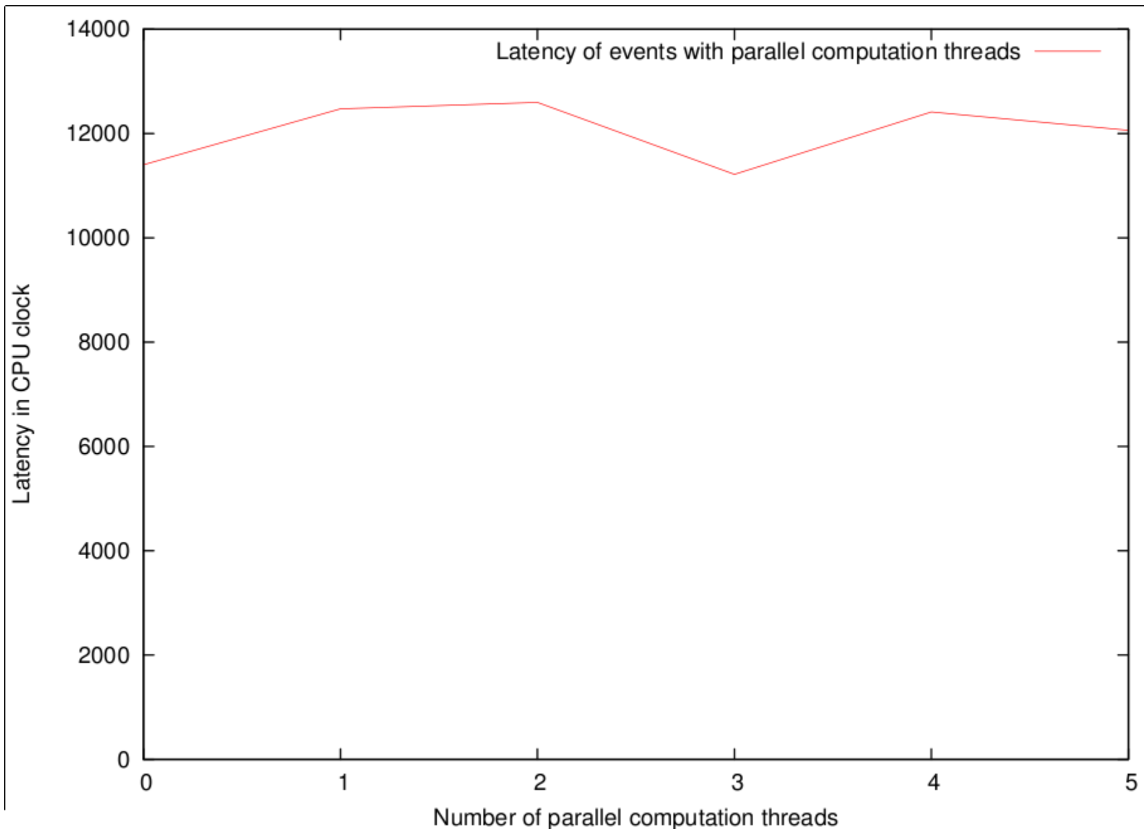
- vCPU syscall around 100 times slower than native
  - Several invocations of ptrace
  - Address space switches

Computation Overhead

	Time in ms	Relation
vCPU	13.733	100%
native	13.643	99,3%

- Compute Fibonacci numbers
- Native performance for compute bound tasks

Event Latency



- Event latency does not depend on number of computational tasks
- Latency around 10,7µs

Conclusion & Outlook

Conclusion

- Low complexity implementation
  - Around 4.000 SLOC
- Low porting effort for legacy applications
  - Ported libav (former ffmpeg)
- Low latency and low overhead
  - Usable for multimedia applications

Future Work

- Implement vCPU on other platforms
- Investigate platforms with native vCPU implementation
  - Microkernel
- Reduce ptrace overhead
  - Use seccomp?
- Investigate effort for legacy applications

Thank you!

Questions?